Generating transportation networks with evolutionary algorithm and swarm optimization

Monday 6th March, 2023 - 09:05

Jason Billard
University of Luxembourg
Email: jason.billard.001@student.uni.lu

This report has been produced under the supervision of:
Grégoire Danoy
University of Luxembourg
Email: gregoire.danoy@uni.lu

Abstract

This report presents a sensitivity analysis of a genetic algorithm used to identify high population density areas in a population density map and a program that simulates a network connecting these areas by modeling the behavior of slime mold. The sensitivity analysis investigates the impact of different parameter values on the performance of the genetic algorithm and identifies the optimal parameter settings to maximise the effectiveness of findind well spaced points with high density population areas. The slime mold simulation receives the point selected by the genetic algorithm as input and uses a swarm algorithm to find the optimal connections between the identified high density areas, potentially representing a rail transportation network. This work contributes to the understanding of genetic algorithm performance and provides a tool for urban planning and analysis. This report is written as part of the third semester of the BiCS program's Bachelor Semester Project course.

1. Introduction

This BSP report presents a sensitivity analysis of a genetic algorithm that finds areas of high population in a population density map. It also presents a program that takes as input a population density map and produces a network connecting the areas of high population, selected using the method described and analysed in the scientific deliverable, by simulating slime mold.

Identifying high population density areas in a population density map is a crucial task in urban planning and analysis. It can inform decisions about resource allocation, infrastructure development, and policy making. Identifying such areas can be a difficult task, when additional parameters are present, such as trying to maximise the distance between the points. This motivates the reason for the development an analysis of such a genetic algorithm.

Genetic algorithms are a common method for solving optimization problems. They utilize principles of natural selection and evolution to search for the optimal solution to a problem. However, the performance of genetic algorithms can be sensitive to various factors, such as the choice of parameters influencing the genetic algorithm, and the structure of the fitness function. The scientific part of this report investigates the impact of different parameter settings on the performance of the algorithm and identifies the optimal parameter values.

In addition to the sensitivity analysis, this paper also presents a program that takes as input a population density map and finds high density area locations. Following this step, the program then connects these key points optimally by simulating slime mold. Slime molds are simple organisms that can find the shortest path between two points by growing and retracting their tubular network of cells. This behavior is simulated using a swarm algorithm and adapted to the problem of connecting the high population density centers.

Overall, this paper contributes to the understanding of the performance of genetic algorithms and provides a program to simulate a network connecting the main population hubs on a map, which could represent a potential rail transportation network.

2. Project description

2.1. Domains

2.1.1. Scientific. The domains of the scientific deliverable are problem modelisation and sensitivity analysis. The first domain consists in modeling a real life transport network problem in a formal mathematical way. The second domain studies how the uncertainties in the parameters affect the uncertainty of the output of functions.

2.1.2. Technical. In the technical part, the most relevant domains are optimization approaches. In this domain, ways are searched to find the most optimal solutions of certain problems using different means. Particularly in this BSP, genetic algorithms and artificial life are the sub-domains considered to solve a certain problem.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. The scientific deliverable will be divided in two parts. The first part will describe how a problem is modeled in a formal way. The problem consists in positioning a specific number of points on a population density map, in a way to maximize the population within a certain radius of the points, with the constraint that points too close together cause a penalty. This part will produce a function, which computes the fitness of any solution. This function will be used in the scientific deliverable to compute the fitness of the population of the genetic algorithm.

The second part of the scientific deliverable will answer the following question: How do the genetic algorithms parameters, more precisely, the size of the population, the amount of generations performed, the crossover and the mutation rate, affect the results? To answer this question, a sensitivity analysis on these parameters will be done, using as model to optimize, the genetic algorithm (GA) problem described in the first part.
To perform the sensitivity analysis, the GA will be run for a constant time, with different parameters each time. The change of the results in each run is compared to the change in the parameters. This information is assessed and visualized, to answer the scientific question.

2.2.2. Technical deliverables. The technical deliverable developed as part of this BSP will be a program receiving a population density map, and generates an optimized transportation network on top of it. Java will be used as a programming language, and Java Swing will be used to create the interface.
The population density map will be given as an image, where darker pixels represent higher density of population. The program will then put nodes in an optimal manner such that they are around high population density areas. Each node has a radius in which the population can reach it, and the program will search the positions of nodes to cover the largest population with the certain amount of nodes. This optimization will be done with a genetic algorithm.
To generate the aforementioned network, a mushroom called slime mold will be simulated. This mushroom is known to generate very efficient networks between food sources, a property that is used in this BSP to find an efficient network between the different nodes. The nodes will be used as virtual food sources with the nodes reaching a larger population as larger food sources.
The parameters for the genetic algorithm and the slime mold simulation can be changed using the user interface.
The produced network will be shown as an overlay on top of the population density map.
In the technical deliverable, the reasoning behind the architecture of the program will be explained. The process of development of this program, as well as the concrete implementation of some algorithm will be described.
A possible extension would be to take into account the topography of the map. Routes should try to reduce their height gradient, as they do in real life. This would further increase the realism of the network.

3. Pre-requisites

3.1. Scientific pre-requisites

3.2. Technical pre-requisites

It is required to have an understanding of the principle behind genetic algorithms to understand the working of the technical park. It is also recommended to understand Java, and have good programming fundamentals. An understanding of kernels in image-processing is also required.

4. A Scientific Deliverable

4.1. Requirements

The requirements of the scientific deliverable consists in two parts. The first part consists in providing a mathematical model, formally describing an optimisation problem, and a way to evaluate solutions to the latter.
The second part consists in analysing the effect of certain parameters on an optimisation approach used to tackle the problem, in this case a genetic algorithm. The algorithm will try to find optimal solutions of the problem described above. The sensitivity of the parameters of the genetic algorithm are computer as part of this deliverable.

The problem that is modeled is described as the following; A population density map is given. The population map is a 2 dimensional grid containing a number in each cell, representing the population at that point on the grid. The goal is to position an exact amount of point, in a way to maximize the population in the proximity of each points. The proximity is defined by some radius.
To not have all the points clumped together, there is a large penalty for points close together, which reduces with the distance.
A solution consists of a set of points, and the fitness of this solution is its population reach minus the penalty. A function which takes this set of points and computes the fitness is produced in this deliverable.
For the sensitivity analysis, the following parameters will be considered: the population size, the number of generations, the crossover rate and finally the mutation rate.
The sensitivity analysis computes how much a change in these parameter affect the influence of the result produced. Here, the result produced are the results produced by a genetic algorithm over the previously modeled problem.

The results of the simulated genetic algorithms should be displayed in graphs, and the findings about the sensitivity analysis should be explained meaningfully.

4.2. Design

4.2.1. Modeling the problem. To model the problem, the different objects are represented mathematically. The population densitiy map is viewed as a matrix $d \in R^{w \times h}$, with w being the width of the map and h the height. The entries $d_{i,j} \geq 0$ is the population at tha point on the map.

Solutions are written as S_p. These are sets of points, with $|S_p| = n$, where n is the number of points in each solution. Each point p has the coordinates (p_x, p_y).

The constant r represents the radius around the points. For the solutions, the population reach of a point p is defined by the population inside the circle defined by the point p and radius r. For this, the function $R(p)$ is defined, which computes the reach of that point (the population inside the circle of center p and radius r).

$$R(p) = \sum_{i=0}^{w} \sum_{j=0}^{h} \begin{cases} d_{i,j}, & \text{if } \sqrt{(p_x - i)^2 + (p_y + j)^2} \leq r \\ 0, & \text{otherwise} \end{cases}$$

To compute this reach, all the elements in the matrix are looped over, and the distance between this cell and the point checked. If cell is inside the circle, the population on this cell is added to the reach.

The penalty of two points is inversely proportional to the distance between them. The function $P(S_p)$ computes the penalty of an entire solution. This means the sum of the penalty between each point.

$$P(S_p) = \frac{1}{2} \sum_{p \in S_p} \sum_{q \in S_p} \begin{cases} \frac{1}{\sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} + 1}, & \text{if } p \neq q \\ 0, & \text{otherwise} \end{cases}$$

Finally, the function $f_p(S_p)$ evaluates how good a solution is. The constant w is a weight, with which we can balance the importance between the population covered and the penalty applied.

$$f_p(S_p) = \sum_{p \in S_p} R(p) - w \times P(S_p)$$

4.2.2. Sensitivity analysis.

4.2.2.1. Sensitivity. The sensitivity of some parameter in a function corresponds to how much a change a change of the parameter affects the result. There are two different sensitivities that can be considered; the local sensitivity, which corresponds to the sensitivity of a parameter for some specific value set for each parameter. The sensitivity for the same parameter may be different, of the other parameters are set to different values.

The global sensitivity of a parameter is the sensitivity to be expected, without knowing any parameters. In this BSP, only the local sensitivity is considered, and the sensitivity depending on the values of different parameters will be shown and explained.

As mentioned previously, the sensitivity s_g of a parameter g_i in a function $F(g_0, ..., g_i, ..., g_I)$ with I parameters corresponds to the change of the function when the parameter is changed. To compute this, the deterministic method for local sensitivity analysis from [1] exists:

$$s_{gi} = \left\{ \frac{\delta F}{\delta g_i} \right\}_{g^0} = \frac{F(g_0^0, ..., g_i + \delta g_i, ..., g_I^0) - F(g_0^0, ..., g_i, ..., g_I^0)}{\delta g_i}$$

In this BSP, the function that will be analysed is a function, representing a genetic algorithm. The function has 4 parameters:

- the population size
- the number of generations
- the crossover rate
- the mutation rate

The result of the function will be the fitness of the best solution of the previously mentioned problem that the genetic algorithm will produce with these specific parameters.

4.2.2.2. Mathematical Model of Genetic Algorithm. A genetic algorithm is a methaheuristic technique to find good solutions for problems, without finding the mathematical best. The intuition behind its functioning is to start with a random set of agents, a population, where every agent has a gene that corresponds to a possible solution. A new population is made, where agents are selected and added or crossed depending on the crossover rate. Some agents are then mutated, and the result is a new set of agents that are a little better, representing the population at the next generation. These steps correspond to 1 generation, and the genetic algorithm repeats this a number of times.

To model genetic algorithm mathematically, a few variables are set:

- P_n is the population, a set of agents a of the n-th generation
- a is an agent, more precisely a sequence of values that represent the gene of the agent, and a_n is the value of the n-th element in this sequence. The size of a gene is $|a|$.
- $f(a)$ is the function that computes the fitness of the agent a

As explained previously, the first generation P_0 is a set agents with random genes. The size of this set $p = |P|$ is called the population size.

Let's create a function $G(P)$, which generates a new population by selecting, crossing and mutating them. The recursive sequence P_n can then be defined as:

$$P_n = G(P_{n-1})$$

The process of selection corresponds to taking an agent of the population depending on its fitness. Many form of selections exist, but a popular one is tournament selection. In that selection heuristic, two random agents are taken from the population uniformly, independent of their fitness. Following this, their fitness are then compared, and the agent with the highest fitness is then selected. Note that the notation $\overset{\text{R}}{\leftarrow}$ corresponds to take a random element of a set with uniform probability.

Take $a \overset{\text{R}}{\leftarrow} P, b \overset{\text{R}}{\leftarrow} P$ on each function call:

$$S(P) = \begin{cases} a, & \text{if } f(a) > f(b) \\ b, & \text{otherwise} \end{cases}$$

Now that the process of selection has been defined, the process of crossing two agents has to be defined. To cross the genes of two agents, an index of the gene sequence is chosen. Everything before that point is taken from one of the parent solution, and the rest from the other. (Note that there are many methods to crossover genes). The following function produces a new crossed-over gene from two different genes:

$$cr(a, a') = (a_n)^\frown(a'_m),$$
$$\text{for } n \in [0, k] \subset \mathbb{N}, m \in [k, |s|] \subset \mathbb{N} \text{ and } k \overset{R}{\leftarrow}\in [0, |a|] \subset \mathbb{N}$$

Generating a new population from the old one consists in first choosing whether a crossed agent will be added, or an agent already present in the previous generation. If a crossed agent is added, first the two parents are selected with the previously defined function. Then the agent generated by crossing these two individual is added in the new population. In the case that the agent should not be crossed, an agent is selected using $S(P)$, and is added as is in the new population. The new population, which will still require the mutation step, now consists of crossed individuals and individuals from the previous generation. The value that defines the ratio between the crossed agents from the entire population is the crossover rate c. This new set will have the same size as the previous generation: $|C(P)| = p$.

$$C(P) = \left\{ \begin{cases} cr(S(P), S(P)) & , h \overset{R}{\leftarrow} [0,1], h < c \\ S(P) & , \text{else} \end{cases} \right\}$$

After generating this new set, the gene of each agent has a probability to be mutated; this means that for each value in the solution, there is a probability of $m_u \in [0, 1]$ that this value is set to a changed by some random value dm, representing a mutation. The set of values that the dm can take is D_m The following function mutates a single value of a gene:

$$m_g(a_n) = \begin{cases} a_n + dm, dm \overset{R}{\leftarrow} D_m, & h \overset{R}{\leftarrow} [0,1], h < m_u \\ a_n, & else \end{cases}$$

Mutating an entire gene corresponds to mutating every value of the gene sequence:

$$m_a(a) = (m_g(a_n)), \text{ for } n \in [0, |a|]$$

Finally, mutating a agent set P:

$$M(P) = \{m_a(a) | a \in P\}$$

Now that the three functions, the selection $S(P)$, the crossover $C(P)$ and the mutation $M(P)$, have been formulated, the generation function can be written down, as the following:

$$G(P) = M(C(P))$$

This is the mathematical modelisation of the genetic algorithm, which is used in the sensitivity analysis.

4.2.2.3. Function to compute sensitivity analysis over. Let's set the function $F(p, c, m_u, n)$, which will be the function that will be analysed. This function computes the population P_n of the n-th generation, with a population size of p, a crossover rate of c and a mutation rate of m_u. The genetic algorithm tries to optimise the previously explained problem, thus it is necessary to find a way to create a sequence that can represent one of the solutions of the that problem. This is known as encoding a solution into a gene. The function $f_p(S_p)$ takes a set of points as input. Lets create a function that converts a sequence to a set of points:

$$p_a(a) = \left\{ (a_{2n}, a_{2n+1}) | n \in \left[0, \frac{|a|}{2}\right] \subset \mathbb{N} \right\}$$

The fitness function then tries to optimise the problem modeled in the first section of the design part, but with the genes decoded so that it can be taken as input:

$$f(a) = f_p(p_a(a))$$

This scientific deliverable, computes the sensitivity of the 4 parameters, over this function.

4.3. Production

When calculating the sensitivity of the different parameters, the other parameters are set to some nominal values. When not indicated, these parameters thus have these values:

- Population size: 100
- Number of generations run: 1000
- Crossover rate: 0.8
- Mutation rate: 0.03

The environment in which the genetic algorithm is running also has some parameters that will stay fixed except indicated otherwise:

- Number of nodes: 20
- The radius (in pixels), defining the reach of each node: 20

- The weight of the penalty given the proximity of the nodes: 40000
- The density map image, on which the nodes are placed:

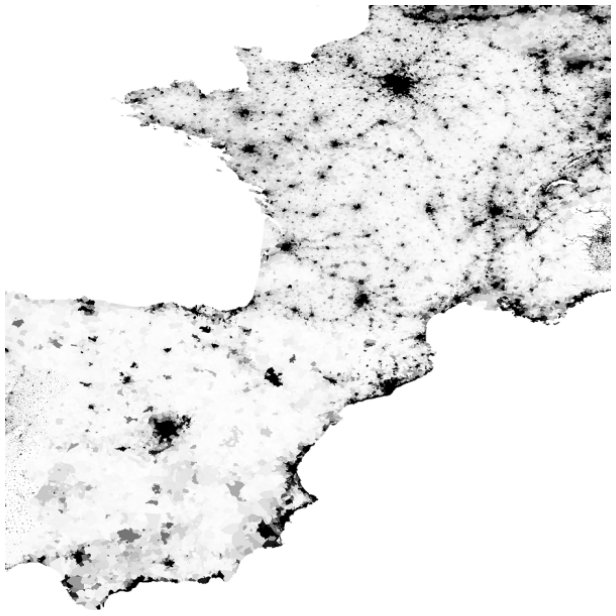


Fig. 1: Gray scale image visualising the population density of France and Spain. The data was collected from [2].

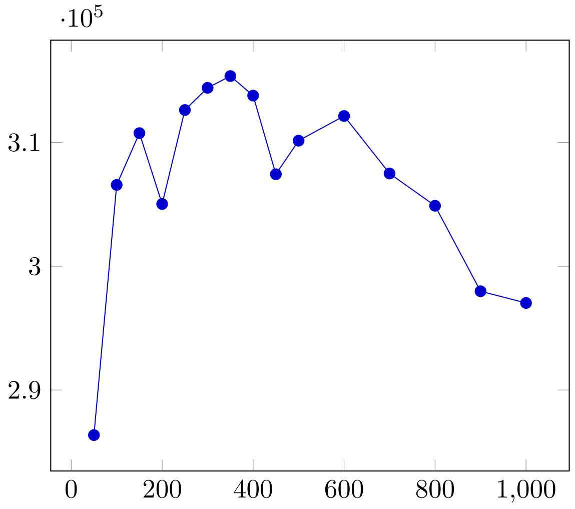
To compute the sensitivity of a parameter, the previously mentioned formula is considered:

$$s_{gi} = \left\{ \frac{\delta F}{\delta g_i} \right\}_{g^0} = \frac{F(g_0^0, ..., g_i + \delta g_i, ..., g_I^0) - F(g_0^0, ..., g_i, ..., g_I^0)}{\delta g_i}$$

This formula represents the derivative of the function that is analysed. Thus, the function will be graphed, and the sensitivity can be deduced by looking at the slope between different points.

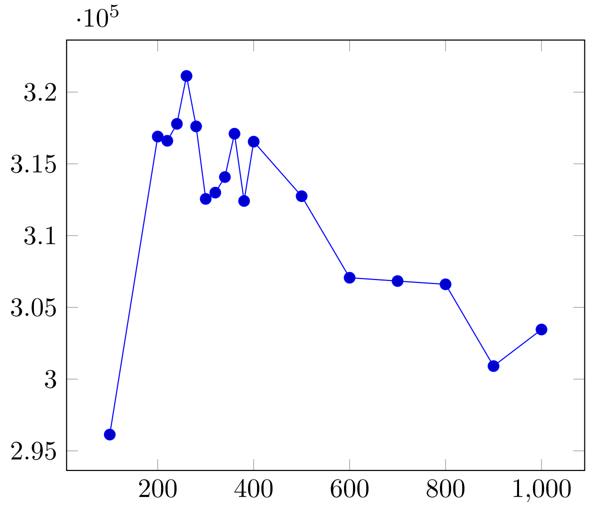
It is important to notice that genetic algorithm are based on random choices; the start population is generated randomly, the selection is picked at random, the mutations are random, and so on. This makes it necessary to take an average of multiple tries to compute a representative result. For this report, the sample count when computing is a value is set to 50, to have a balance between correctness and a reasonable computation time for each result. In this report, it was decided that the average of the maximum fitness of the last population is the output of the function, which will be referred to as the score of the function for simplicity reasons. Another possibility would have been to take the median, to represent the average fitness of the population. However, in the context of this problem, it is more likely that only one solution will be kept, which will be the maximum of the population.

4.3.0.1. Population count. The following graph contains the data computed, by changing the population count of the genetic algorithm from 50 up to 1000, increasing by 50 then 1000 every step. If the population would just increase, without decreasing the number of generation counted at the same time, there would be a necessary constant increase, and more actions are taken in general, and a larger space of solution is explored. To compensate this fact, when analysing the population count and the number of generation, the total number of operations will be kept constant. In each generation, one agent represents one operation. Thus, the value $p \times n$, where n represent the number of generations calculated, is a constant. The following graph shows the only the population change, but remember the constraint set previously.



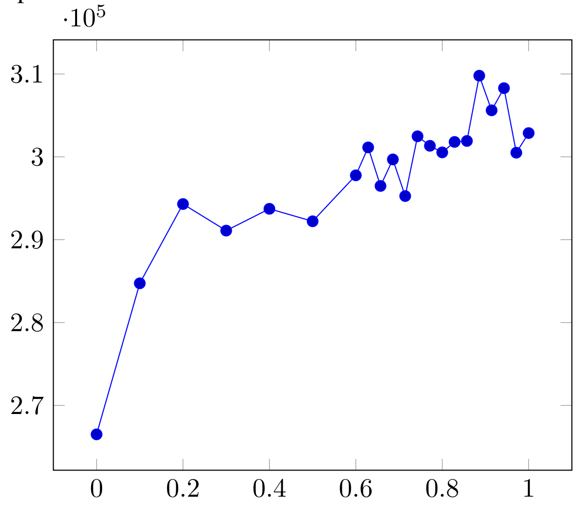
The results computed show a graph, first increasing up to some maximum, then decreasing again. This indicates with some population sizes, a larger space of solutions is explored, and better scores are computed with the same amount of operations. Using a very low population may mean that a large number of generations will be computed, but it remains that never more than p solutions can be explored and crossed at the same time. This means that the local maximum will be reached due to the large number of generations, but since the locality is very small, it will not produce a good result. With a very large population, it may be possible to explore a larger number of solutions in one generation, but since only very few generations are performed. In the solutions space, this can be seen as a very large locality being explored, but since the number of generations is so low, the maximum on this locality will not be reached.
The peak is reached with a population of 350. It is interesting to consider the previously mentioned constant, being $p \times n$. This constant is 100000 in this evaluation (since the default population is 100, and the default generation count is 1000). Computing the square root of this constant gives a number that will be rounded to 316, which is close to the population at which the maximum value was obtained.

4.3.0.2. Number of generation. The same procedure is done with the number of generation. This time increasing the number of generations computed from 100 to 1000 by steps of 100. The same constraint is applied as during the analysis of the population size.



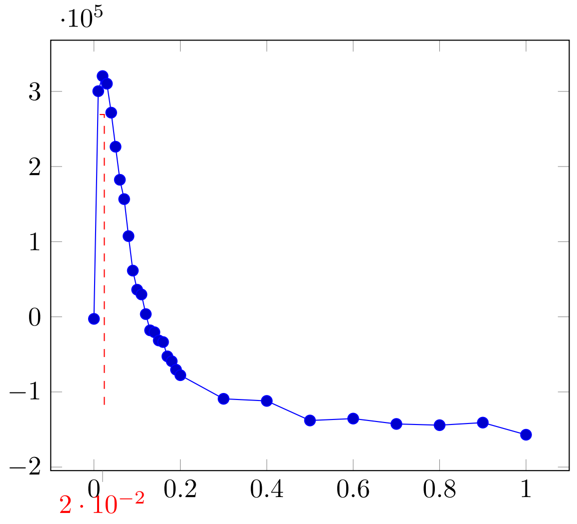
By increasing the generation, a similar result as with the population size comes out, as the values of one could be used for the other one. The only change is what the x-axis contains. However new values have been calculated with a sample count of 100, however the result seem to vary more. The explanations of the results for this rising, then decreasing graph are the same as what has been explained in the analysis of the population size. A very low number of generations, means that the population will be very large, but very few generations will be performed, many places on the solution space will be explored, but the lack of generations means that the result does not have time to converge. Incidentally, a very large generation count, means that only a very small population will be evaluated. This, the local maximum will be found but these are very localised.

4.3.0.3. Crossover Rate. As a reminder, the crossover rate specifies the ratio of crossed agents to the entire population. The crossover rate is graphed from 0.1 to 0.9, with steps of 0.1 and 0.05 at the higher values. Due to the high noise, a sample rate of 100 was used for these values.



The outcome of the graph shows a general trends demonstrating that higher crossover rates are better. A crossover rate of 0 means that no two agents are ever combined, highly reducing the solutions explored. However with a crossover rate of 1, every agent in each generation comes from two previous agents, increasing the diversity highly. However, with a crossover rate of 1, very good solutions aren't kept intact, and are always mixed with some other solutions, which makes it hard to explore around the best solutions. On the graph, we notice an increase until around 0.9, which the drops before reaching 1.

4.3.0.4. Mutation Rate. The mutation rate corresponds to the probability of an allele in the gene of an agent to receive a mutation. In the mathematical definition of the genetic algorithm, a random value is added to the mutated allele. That value dm is taken uniformly from a range of value D_m. In this context, this value is set to $\left[-\frac{1}{6}min(w,h), \frac{1}{6}min(w,h)\right]$, where w and h are the width and height of the population density map. In this graph, the mutation rate takes the value from 0 to 1, but with smaller step sizes on values less than 0.2 and larger steps on the rest of the range.



The results show a very high peak at a specific value, and very low scores on the extremes 0 and 1. Incidentally, the range of the y axis compared to the other graphs is much larger. This demonstrates that the mutation rate has a very large influence on the results produced, and using the optimal value is very important. At a mutation rate of 0, the only solutions that are explored, are the possible crossovers from the original randomised population. No new solutions are explored, since a mutation rate of 0 restraints the genetic algorithm from it's stochastic ascend towards the local maximum. This results in a very low score.

For large mutation rates, every agent will be mutated many times, and they will not be able to stabilise to some optimal local maximum solution. A mutation rate of 1 means that every allele is mutated for every agent every generation. This results in fully randomized agents, every generation, which will result in a very low score, even lower than with a mutation rate of 0.

The maximum at 0.02 correlates precisely to the heuristic to set the mutation rate to $\frac{1}{|a|}$. In our case, the allele size is 40 (there are 20 nodes, each having an x and y coordinate), thus the heuristic would set the mutation rate to 0.025.

4.4. Assessment

The goal of the scientific part was to find the sensitivity of the different parameters. Now that the function corresponding to the change of these parameter has been computed, it is possible to look at the derivative of these graphs, and determine the sensitivity of these parameters.
By looking at the slope of the function on a certain x-value, it is possible for us to find out two facts:

- the sensitivity of the parameter at a specific value. This will correspond to the absolute value of the slope. If the slope goes down or goes up a lot, it signifies that the sensitivity is very high around this value of the parameter.
- Whether the output would become higher with or lower when increasing or decreasing the value. This corresponds to whether the slope goes up or down.

By looking at the graphs generated for each parameter, there is a similarity for all of them. Each graph contains a peak, corresponding to the value for the parameter at which the results will be optimal.

The goal of sensitivity analysis is to find the best parameters to run the genetic algorithm with. Using the previously computed graphs, we can find the optimal parameters.

Using the graph computed for the population size, an optimal value of the population can be found. In this graph; 350 agents. As mentioned previously, a constraint was set, that the number of evaluations of the agents stays constant. Since the number of agents is 350 and that constant is 100000, a division can give us the ideal number of generation. Using these values, the optimal number of generations obtained is 285. This value corresponds precisely with the peak obtained on the second graph, containing the variation of the number of generation. This demonstrates the consistency of the calculated results.

For the crossover rate, the value of 0.88 can be read of the graph, and for the mutation, the graph contains very little noise and the optimal value 0.02 is obtained.

Overall the optimal parameters are the following:

- Population size: 350
- Crossover rate: 0.88
- Mutation rate: 0.02

Using the optimal values obtained, this genetic algorithm can now be run optimally. The main goal of the sensitivity analysis has been achieved. Here, two results can be seen produced with the optimal parameters on 1000 generations.

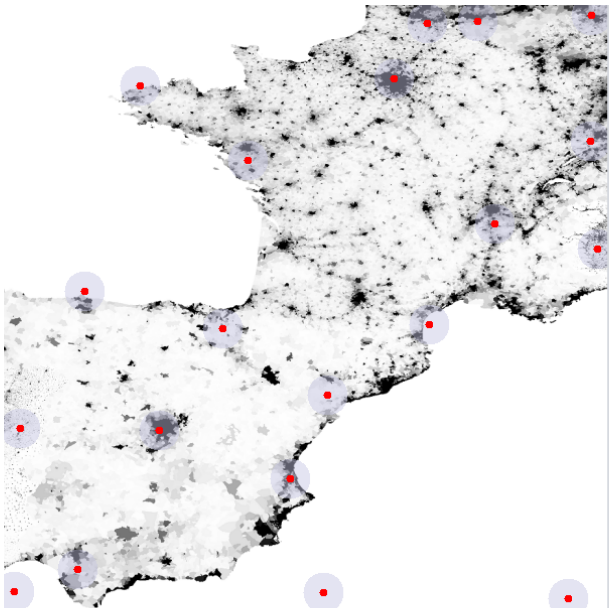


Fig. 2: Image the showing points selected by the genetic algorithm on population density map. The proximity penalty is 40000.

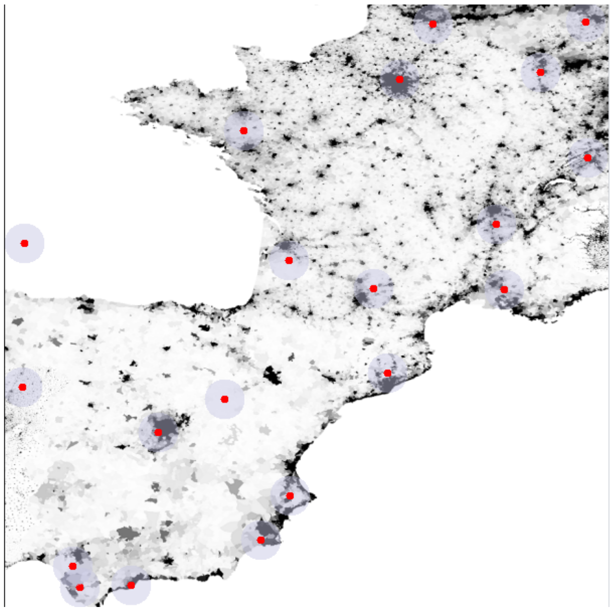


Fig. 3: The same type of image with a proximity penalty is 20000.

We notice that a few points try to go on the edges of the image to reduce the proximity penalty.

5. Program to generate transportation networks with evolutionary algorithm and swarm optimization

5.1. Requirements

The requirements of the technical deliverable consists in developing an interactive software that produces a transportation network on top of a population density map. The population density map is given as an image, where darker pixel mean higher population.

To generate this transportation network, the software uses a combination of genetic algorithm and slime mold simulation. The first step consists of positioning a number of points, defined by the user, in a way that each point has a highest possible population within some radius (this radius is also adjustable by the user). However, these points have a penalty dependant on the distance between them. Refer to the modelisation of the problem in the scientific part for a precise mathematical description of the problem at hand.

After selecting these nodes, the resulting points are pipelined towards the slime mold simulation. More precisely, they are treated as food sources, on which this slime mold will develop and connect. The simulation of this slime mold uses a swarm approach, in which a large number of agents perform defined actions, and together they manage to optimise a solution. Parameters such as the number of agents and other parameters of the slime mold simulation can all be adjusted by the user. After processing the simulation for a amount of iterations specified by the user, the network made by the slime mold can be exported to an image.

5.2. Design

A pipeline is established in accordance with the specifications in order to produce the required program. Understanding the actions that must be completed at each stage of the program is necessary to determine the pipeline. The first part of the program consists in positioning points on a population density map so that they cover the most population whilst keeping a distance with each other. This is done with a genetic algorithm that finds an agent with the best fitness score, which represents the quality of the solution. The points selected with this step are then given to the next step, which uses these points as simulated food sources for a slime mold. This virtual slime mold is simulated, and creates a network connecting these selected points.

The following picture contains an general overview, describing the pipeline of the program, and the underlying modules, which perform the different steps.

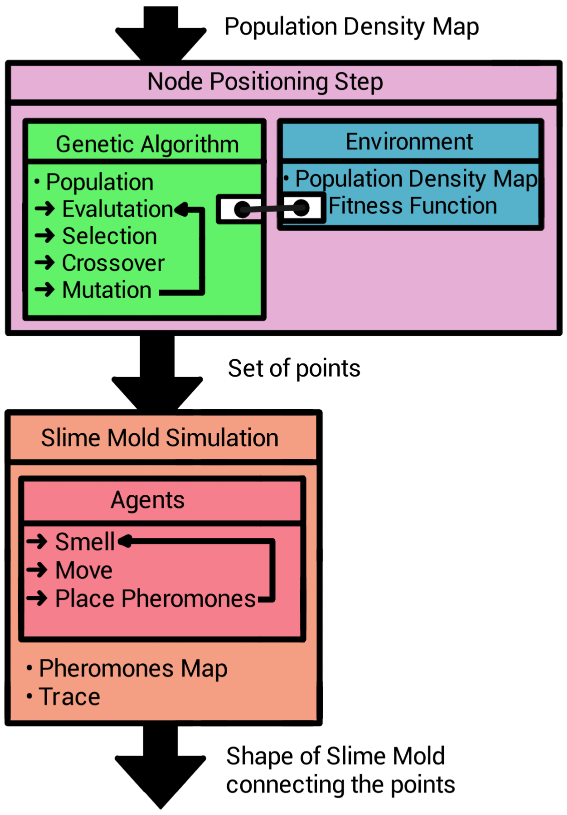


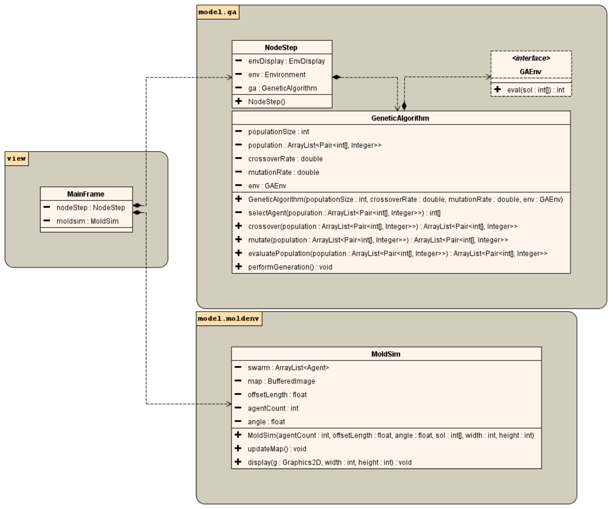
Fig. 4

As can be seen in the pipeline, and also mentioned previously, the program consists of two distinct parts, which is reflected in architecture of the program.

5.2.1. Overview of architecture of the program. As this program is developed in Java, the program is devised in classes, having their own functions. One of the requirements is that this program has a user interface, which will be the first class *Mainframe.java*.
As this program uses a genetic algorithm, a class containing the implementation of this algorithm is also necessary (*GeneticAlgorithm.java*). Here a new genetic algorithm can be run on an arbitrary environment. The specific implementation of the selection, mutation and crossover correspond to the model described in the scientific part. A simple explanation of the algorithm is [5], but more details can be found in [6].
For a genetic algorithm to work, it needs to evaluate its agents. Thus there exists a fitness functions, which computes the quality of an agent in the population of the genetic algorithm. In this case, the fitness function is inside an *Environment.java* class, which represents the environment that the population exists in. In this program, the environment consists of the density map image, and the fitness is calculated as mentioned in the scientific part. This environment models the problem described there.

The second step consists in simulating a slime mold with the previously selected points. Slime molds, known scientifically as Physarum polycephalum, are a type of mushroom that are famous for their interesting shape. When given multiple food sources, they find an optimal path connecting these sources. They have been used previously to create an alternative Tokyo subway system [3]. Their characterstic behaviour have been tried to be simulated virtually, and such a simulation, using swarms of agents (page 6 of [4]), will be used in this semester project. This simulation will be contained in the class *MoldSim.java*.

Here follows a simplified UML of the program:


model.ga
NodeStep
− envDisplay : EnvDisplay
− env : Environment
− ga : GeneticAlgorithm
+ NodeStep()
<interface>
GAEnv
+ eval(sol : int[]) : int
GeneticAlgorithm
− populationSize : int
− population : ArrayList<Pair<int[], Integer>>
− crossoverRate : double
− mutationRate : double
− env : GAEnv
+ GeneticAlgorithm(populationSize : int, crossoverRate : double, mutationRate : double, env : GAEnv)
− selectAgent(population : ArrayList<Pair<int[], Integer>>) : int[]
+ crossover(population : ArrayList<Pair<int[], Integer>>) : ArrayList<Pair<int[], Integer>>
+ mutate(population : ArrayList<Pair<int[], Integer>>) : ArrayList<Pair<int[], Integer>>
+ evaluatePopulation(population : ArrayList<Pair<int[], Integer>>) : ArrayList<Pair<int[], Integer>>
+ performGeneration() : void
view
MainFrame
− nodeStep : NodeStep
− moldsim : MoldSim
model.moldenv
MoldSim
− swarm : ArrayList<Agent>
− map : BufferedImage
− offsetLength : float
− agentCount : int
− angle : float
+ MoldSim(agentCount : int, offsetLength : float, angle : float, sol : int[], width : int, height : int)
+ updateMap() : void
+ display(g : Graphics2D, width : int, height : int) : void

To develop the program, a basic framework or skeleton of the program was first created. Then, upgrades were gradually added. This provided a solid foundation upon which additional features and functionality could build. This approach to building a program has the advantage of allowing each block of code to be thoroughly tested, making it easier to pinpoint any bugs that may occur. In addition, some elements of the program have first been prototyped using a programming language called Processing. Once a working prototype was achieved in Processing, the functionality was replicated in the main Java project.

5.3. Production

To describe the development process of the technical part of this semester project, the algorithms and the code of the important features of the program will be explained in depth.

5.3.1. Genetic Algorithm. The first block that has been programmed was the genetic algorithm. It was later reviewed due to a bug discovered when looking at results produced for the scientific part. As explained in section 4.2.2.2., the genetic algorithm consists in repeatedly improving a population every generation. The algorithm performing a series of step every generation, which results in a better population. It selects agents based on their fitness score and may cross them with another chosen agent depending on the parameter `crossoverRate`. After selecting a number of agents such that the current population size is equal the that of the last generation, the program mutates the agents based on a probability of `mutationRate`. Here is the function that performs one generation:

```java
public void performGeneration() {
    ArrayList<Pair<int[], Integer>> pop = crossover(population);
    pop = mutate(pop);
    pop = evaluatePopulation(pop);
    pop.sort((p1, p2) -> p2.b - p1.b);
    genCounter++;
    population = pop;
}
```

First the function generates a new set of agents, `pop`, using the `crossover` function, which will later be explained in detail. This function selects agents and may cross them with other agents. This new set of agent is then passed through the `mutation` function, which mutates the agents with some probability. The agents are then evaluated, and then sorted according to their fitness. The sorting is necessary to display the information such as the maximum fitness, the median agent and more, but it is not necessary for the functioning of the genetic algorithm. Finally, the generation counter is increased and the new set is assigned to the current population variable `population`. The population is stored as a list of pair, where each pair contains the agent itself `p.a`, and its corresponding fitness `p.b`.

The function `crossover` generates a new population from a current population by adding a new or a crossed agent repeatedly. Here is the code perfoming this task (the type of the parameter has been ommited for readability):

```java
ArrayList<Pair<int[], Integer>> crossover(population) {
    ArrayList<Pair<int[], Integer>> crossed = new ArrayList<>();
    for(int i = 0; i < populationSize; i++) {
        if(Math.random() < crossoverRate) {
            int[] parent1 = selectAgent(population);
            int[] parent2 = selectAgent(population);
            int[] crossedAgent = crossAgent(parent1, parent2);
            crossed.add(new Pair(crossedAgent, 0));
        } else {
            crossed.add(new Pair(selectAgent(population), 0));
        }
    }
    return crossed;
}
```

The new population, called `crossed`, within the funcion, is populated in the loop. Here, the parameter `crossoverRate` influences the ratio of crossed to non-crossed agent in the new population. To do this, some random number between 0 and 1 is chosen. This number has a `crossoverRate` probability to be smaller than `crossoverRate`, which is why we can use this condition to perform the crossing operation with the desired probability. If crossed, two parents are first selected with `selectAgent` (which will also be explained later), and then these parents are crossed with `crossAgent` (explained after `selectAgent`), resulting in a new agent. When inserting these new agents, whether crossed or not, the fitness value is set to 0 since they have not been evaluated. It also does not make sense to evaluate them at this point since they will later be mutated, making the fitness value computed here irrelevant. Finally, the new set is returned.

Here is a closer look at the `selectAgent` function:

```java
private int[] selectAgent(ArrayList<Pair<int[], Integer>> population) {
    Pair<int[],Integer> p1 = population.get((int)(Math.random() * populationSize));
    Pair<int[],Integer> p2 = population.get((int)(Math.random() * populationSize));
    if(p1.b > p2.b) return p1.a.clone();
    return p2.a.clone();
}
```

There are many ways in the literature of genetic algorithms to select agents from the population. A small selection of these include roulette wheel selection, where each agent is assigned a probability of being selected based on its fitness; rank selection, where agents are ranked according to their fitness and a probability is assigned to each rank; tournament selection, where a small number of agents compete against each other and the highest-fitness agent is selected. This last method is the one used in this BSP. The tournament is done between two randomly agents selected from the population, selecting the agent with the highest fitness. Here, the two agents are `p1` and `p2`, which are selected randomly from the population. And a copy of the better agent is given, as in Java, objects are passed by reference. A change in the value returned would also affect the original population.

The function `crossAgent`, returns a new agent, given two parent agents. In genetic algorithms, crossover methods are used to create new agents from two parent agents by combining their genes in some way. Some more well known crossover methods are single-point crossover, where a single point is chosen at random and the genes to the left of the point are exchanged between the two parents; two-point crossover, where the genes between two randomly selected points are swapped; uniform crossover, where there is a likelyhood for each gene to be exchanged between the two parents and arithmetic crossover, where the offspring are created by taking a weighted average of the genes of the two parents. This technical deliverable uses the single-point crossover method to cross agents. The code implementing this method is the following:

```java
private int[] crossAgent(int[] parent1, int[] parent2) {
  int[] crossoverSol = new int[env.getSolLen()];
  int crossPoint = (int)(Math.random() * env.getSolLen());
  for(int j = 0; j < env.getSolLen(); j++) {
      if(j >= crossPoint)
    crossoverSol[j] = parent1[j];
      else
    crossoverSol[j] = parent2[j];
  }
    return crossoverSol;
}
```

First, a new array that will contain the offspring is created, then the point, on which the single-point crossover will be performed, is chosen at random. Then a loop iterates over the gene length, and adds the gene of one parent if the index is greater than the selected point, otherwise the gene of the other parent is copied. This new solution is returned.

The function that performs the mutation is the last function explained in this part. This function mutates every agent using the following code:

```java
for(int j = 0; j < env.getSolLen(); j++) {
    if(Math.random() > mutationRate)continue;
    int dev = (int)(Math.random() * maxDev - maxDev / 2);
    population.get(i).a[j] += dev;
}
population.get(i).a = env.correct(population.get(i).a);
```

As has been described mathematically in the scientific part, the mutation gives each gene a likelyhood of `mutationRate` to mutate. In this implementation, the mutation is chosen uniformly from $-maxDev/2$ to $maxDev/2$. An important point is the last line, which sends the agent to the environment. The environment checks whether this solution is allowed, and returns a corrected version. This is done in case the mutation may cause some undesired solution. In this BSP, this was done to not have any points outside the image. Since the points are stored as a list containing the x and y coordinates of each point, it may be that the mutation sets a point outside the image, which is then corrected by this line.

5.3.2. Environment and fitness function. For a genetic algorithm to work, a fitness function is necessary. This function is implemented here in the `Environment`. This class corresponds to the implementation of the model built in the scientific part. It computes the fitness of the solution by summing the population reach of each point of the solution, and then applying some penalty to close points.

To calculate the reach of each point, a matrix containing the reach for each point is computed in advance, such that during the evaluation of the agents, it is not necessary to calculate the reach again, and the lookup time has a complexity of $O(1)$. This code snippet contains the loop, calling the reach function on each point of the population density matrix `map`, and saves this results in a matrix called `processed` which will contain the reach for each point:

```java
private void preprocess() {
    processed = new int[map.length][map[0].length];
    for(int i = 0; i < map.length; i++) {
        for(int j = 0; j < map[0].length; j++) {
            processed[i][j] = reach(j, i);
        }
    }
}
```

The reach function is given the x and y coordinates of a point, and computes the population within a radius `nodeRadius`. It performs this by looping over all rows in which the circle is contained, and sums the value inside the circle.

```java
private int reach(int x, int y) {
int sum = 0;
for(int h = -nodeRadius; h <= nodeRadius; h++) {
    double a = Math.acos((double)h / nodeRadius);
    if(h + y < 0 || h + y >= map.length) continue;
    int dw = (int)(Math.sin(a) * nodeRadius);
    for(int w = x - dw; w <= x + dw; w++) {
            if(w < 0 || w >= map[0].length)continue;
            sum += map[h + y][w];
        }
    }
    return sum;
}
```

The method first declares a variable `sum` and initializes it to 0, this variable will contain the total final reach at the end of the functio call. It then loops from the left of the circle (-`nodeRadius`) to the bottom of the circle. For each value of rows within this circle, it calculates at which angle from the vertical line, the points on the circle on this row are located (see image).

The method then checks whether h + y (the y-coordinate of the points of this row) is less than 0 or greater to the length of the map array, which would mean it is outside the map. If this is the case, it continues to the next row. The function then calculates a value `dw` (see image for more details) by multiplying the sine of the previously calculated angle with the `nodeRadius`. This is then casted to an integer.

Using `x` and `dw`, it is possible to find the range of columns that are in the circle, on one specific row. These range from `x - dw` to `x + dw`. The method then loops in this range with the value `w`. For each value of w, it checks whether w is within the map, and if not, it skips to the next x-coordinate.

The program then adds the value of `map[h + y][w]` to the sum variable, because this point is known to be within the circle.

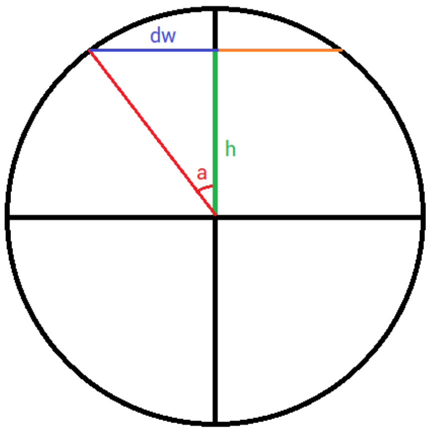


Fig. 5: Image describing the values h, a and dw in the code.

To calculate the fitness of the agents, the following function is used

```
public int eval(int[] sol) {

    int totalReach = 0;
    for(int i = 0; i < numNodes; i++)
        totalReach += processed[sol[i + 1]][sol[i]];

    int totalPenalty = 0;
    for(int i = 0; i < numNodes; i++)
    for(int j = i + 1; j < numNodes; j++)
        totalPenalty += penalty(
        sol[2 * i], sol[2 * i + 1],
        sol[2 * j], sol[2 * j + 1]);

    return totalReach - totalPenalty;
}
```

This function first sums the reach of each points, and saves this sum in the variable `totalReach`. Then, it iterates of possible node pair. This is done by first iterating from `0` to `numNodes` with `i`, then only iterating from `i + 1` to `numNodes`. This removes the possibility of passing through the pair twice, and will not compare two times the same point with each other. For each node, whose coordinates are stored in the `sol` array, representing the genome of the agent as x and y coordinates following each other. Hence the code `sol[2 * k], sol[2 * k + 1]` to retrieve the coordinates of the node `k`. These coordinates are passed to the penalty function, which multiplies the inverse of the distance of these points with a factor `proxPenalty`. The details are described exactly in the scientific part.

After selecting the points with the genetic algorithm, the environment is also responsible to produce in output, which can be send to the next step in the pipeline; the slime mold simulation. This simulation requires a list of points, with their reach attached to it. This is done in a function called `exportForSlimeMold`, which is given an agent, and returns an array that contains for each node, the x and y coordinate, as well as its reach.

5.3.3. Slime Mold Simulation. In this subsection, the decisions made around the implementation of the slime mold simulation will be described. This consists in the limitations encountered, the difficulties of implementing the data points and the reason for the final implementation.

The implementation of the slime mold simulation was inspired by [4]. This is a paper describes how such a simulation could be used to recreate the cosmic web. On the page 6, they define the ”Monte Carlo Physarum Machine”, a detailed implementation of such a simulation using a swarm of agents that deposit pheromones, guiding the rest of the swarm.

The machine is composed of 3 elements. The first element is a swarm of agent, where each agent performs actions independently of the direct actions taken by the other agents. Each agent behaves the same way, which means that if they were in the same conditions, they would take the same action with the same probability, all though they may not take the same action in each instance, since some randomness is involved. The second element is the pheromone field, where the agents deposit pheromones, and also where the agent ”smell” to decide what direction they will move towards. Finally, because of limitations that made it impossible to follow the precise implementation of the model described in the paper, a map containing the points given by the genetic algorithm is also present.

Each time step of the simulation consists of two parts; the propagation step, where each agent in the simulation is moved and deposit some new pheromones; and the relaxation step, where the pheromone field is diffused and attenuated.

A brief explanation of the principle of this simulation is the following. Each agent tries to go in the direction, where more pheromones are present. They do this by ”smelling” at some distance (which can be changed in the interface) away, on the left, and on the right, and turning in that direction. They then deposit some pheromones on the field, and the pheromones propagate through the ”air”. In the paper, the data points, in our case these are the nodes selected by the genetic algorithm weighted with their reach, are inserted as agents that do not move, but deposit an amount of pheromones depending on their weight. The accumulation of these concepts produce a simulation, where the agents tend to go move between data points, and try to maximise their presence near pheromones. How this behaviour is related to the biological *physarum polycephalum* is not explained.

5.3.3.1. Limitations. One of the requirements of this technical deliverable, is that the produced software should be interactive, and the user only should have to wait a reasonable amount of time to produce the desired results. However, the first implementation of this simulation was slow, making a change with compromise necessary.

The first implementation used an two dimensional integer array as pheromone field. The data points could then be inserted with their original reach value. The reason this was necessary, is because in the paper, the number of data points were a few orders of magnitude higher than within this BSP, where it doesn't exceed 30. By inserting these larges values, and after a few steps of propagation, a circle, with decreasing pheromone density the further away from the points appeared. This produced a acceptable result, with the only, but very large, issue being the large amount of time required to compute the results.

Some analysis showed that the main cause of the delay was the relaxation step, where the pheromones are diffused in the field by applying a kernel on each point of the field. This operation is implemented in the java awt class ConvolveOP. Here, images are used instead of two dimensional arrays, so the implementation was changed to use images. The speed was now desirable, but a new issue appeared: the maximum value of a pixel is 255, which was not sufficient to compensate the lack of data points. This is why, at first, a circle of data agents were put around is node. However, this goes against the goal of the first step, which is to find a few points that the slime mold should connect.

The last and current implementation, which will be explained in detail, uses a third image to represent the data points. It was first tried to draw this data point image every time step on the pheromone field, but the accumulation of the pheromones was not correct, so it was decided that the agents would measure the pheromones in both the pheromone field and the fixed precomputed data point pheromone field.

5.3.3.2. Implementation: Agents. Each agent stores 2 values that represent their state; their position vector `pos` and their direction vector `dir`. The agent also have two main functions, being `smell`, which changes the direction of the agent by smelling the pheromones, and `move`, a trivial function which updates the position of the agent by addition its direction vector to its position vector.

The smell function checks the amount of pheromones in 3 different positions:

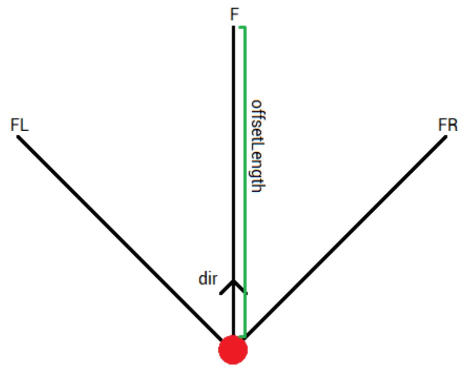


Fig. 6: The 3 positions that the agent smells at. F, FL, FR.

A single parameter exists to adjust the position smelled at. It is the `offsetLength`, which defines how far away the agent will inspect the pheromone field at. It smells once straight in front `F`, with a 45° angle to the left `FL` and finally with a 45° to the right `FR`.

After getting the value of the amount of pheromones on these points, the agent decides in what direction to orientate itself towards. If the value of `F` is greater than both `FL` and `FR`, the agent will not change its direction, since it is already moving towards the location with greatest amount of pheromones. If both `FL` and `FR` have a higher value than the middle, the agent chooses at random in which direction it will go. The two final cases are the following. Either `FL` <`F` <`FR` which means that the value strictly increases from right to left. In that case, the smartest choice to go towards the pheromones is to turn left. Or, `FL` >`F` >`FR`, which is the opposite as the previous case. The value increase from left to right, and the agent will turn right.

```
public void smell() {
    int F, FL, FR;

    Vect offdir = dir.copy().setMag(offsetLength);
    Vect leftdir = offdir.rotate((float) (-Math.PI * .25));
    Vect rightdir = offdir.rotate((float) (Math.PI * .25));
    Vect Fdir = offdir.add(pos);
    Vect FLdir = leftdir.add(pos);
    Vect FRdir = rightdir.add(pos);

    F = map.getRGB((int) (Fdir.x), (int) (Fdir.y))>>24;
    F += dataPointImage.getRGB((int) (Fdir.x), (int) (Fdir.y))>>24;

    FL = map.getRGB((int) (FLdir.x), (int) (FLdir.y))>>24;
    FL += dataPointImage.getRGB((int) (FLdir.x), (int) (FLdir.y))>>24;

    FR = map.getRGB((int) (FRdir.x), (int) (FRdir.y))>>24;
    FR += dataPointImage.getRGB((int) (FRdir.x), (int) (FRdir.y))>>24;

    if (F > FL && F > FR)  return;
    if (F < FL && F < FR) {
        if (Math.random() < .5)
            this.turnLeft();
        else
            this.turnRight();
        return;
    }
    if (FL < FR)
        this.turnRight();
    else if (FR < FL)
        this.turnLeft();
}
```

First, the 3 variables `F`, `FL` and `FR` are declared, which will contain the amount of pheromones on the points. Then, 3 vectors declared, each one of them, containing the coordinates of the points `F`, `FL` and `FR` relative to the position of the agent. The true value of these coordinates are then stored in `Fdir`, `FLdir` and `FRdir` by adding the position of the agent.

For each of these points, the value of the alpha channel is obtained for the pheromone field (variable `map`) as well as the image containing the data points `dataPointImage`. As these values are stored in a `BufferedImage`, the way the value of the alpha channel is retrieved, is by first retrieving the full ARGB colour at this pixel with the function `getRGB`. The colour is represented as an integer, whose bytes describe the followings `0xAARRGGBB`. To retrieve the `0xAA`, a bitshift fo 24 bits to the right is performed. After summing the value of the pheromone field and the data map for the 3 points, they are compared to change the direction accordingly.

It is first checked whether the agent should continue straight forward. If so, the program quits the `smell` function. Otherwise, it checks whether the agent has to take a random choice. In that case, a random number is generated and compared to 0.5, turning left if less and right if greater. The last two cases check if the agent should strictly turn left or right. The `turnLeft` and `turnRight` function change the direction by a amount stored in the variable `angle`, which can be changed interactively with the user interface.

An important note is that the provided code has been simplified for readability. In the implementation, every position is always wrapped around the maximum and minimum value of its range. Meaning that if the agent would smell over the limit of the pheromone map, it would wrap around to the other side of the map. This decision has been taken because of the simplicity of implementation, but also because the size of the field limits the range of motion of the agents, and they would not be able to create the desired shape.

To update each agent, the following function, that makes each agent call `smell` then `move`, exists:

```java
void updateAgents() {
    swarm.stream().parallel().forEach((a) -> {
        a.smell();
        a.move();
    });
}
```

The propagation of the pheromones is done using the following method, which just sets the colour of each pixel of the pheromone map to the weight corresponding to each agent. The weight is the same for each agent, and is an ARGB value.

```java
void propagatePheromones() {
    swarm.stream().parallel().forEach((a) -> {
        map.setRGB(
            (int) a.pos.x,
            (int) a.pos.y,
            a.getWeight()
        );
    });
}
```

5.3.3.3. Implementation: Relaxation step. In the paper used for this part of the technical deliverable, the relaxation step is said to "*ensures that the simulation eventually reaches an equilibrium*" [4]. This is done by spreading the pheromones through the field, like a smell would diffuse in the air. To perform this step, a blurring kernel is applied to the field to spread the values evenly in all directions. To make sure that the pheromones do not accumulate and saturate the field, each value of the pheromone field is reduced in magnitude. This could be done either by multiplying each value of the field with some factor less than 0. The method that has been used in this BSP is be to change the kernel, such that if the kernel is applied, the attenuation is done simultaneously.

The code to define the kernel is the following:

```java
float attenuation;
float[] baseKernelMatrix = {
    1/ 14f, 1 / 7f, 1/ 14f,
    1 / 7f, 1 / 7f, 1 / 7f,
    1/ 14f, 1 / 7f, 1/ 14f,};
float[] diffuseKernelMatrix;
Kernel diffuseKernel;

void setAttenuation(float att) {
    attenuation = att;
    diffuseKernelMatrix = new float[baseKernelMatrix.length
     ];
    for (int i = 0; i < diffuseKernelMatrix.length; i++) {
        diffuseKernelMatrix[i] = baseKernelMatrix[i] *
     attenuation;
    }
    diffuseKernel = new Kernel(3, 3, diffuseKernelMatrix);
}
```

A base kernel `baseKernelMatrix`, defined manually, is used to generate the `diffuseKernel`, the kernel used later to diffuse the image. In the `setAttenuation` function, some factor for the attenuation can be set. The new diffuse kernel then becomes the base kernel, where very value of this kernel has been multiplied by the attenuation factor. The reason why applying this kernel to the image produces the same result as first attenuating the image, then applying the purely-diffuse kernel, is the following. By multiplying each of the values of the kernel with the attenuation factor, it is as if the kernel would be applied on the field, but each pixel of the kernel has been attenuated. If f is the value of the field, a the attenuation factor and k the value of the kernel at some pixel, applying first the attenuation then the kernel, would be represented as $(f \times a) \times k$, but because of associativity, $f \times (a \times k)$ is also valid, which is done in this case.

To apply this kernel, two images (`map` and `newmap`) are used, and switch every frame to reduce the memory allocation and initialisation of a `BufferedImage` time.

```java
void relaxationStep() {
    (new ConvolveOp(diffuseKernel)).filter(map, newmap);
    BufferedImage temp = newmap;
    newmap = map;
    map = temp;
}
```

In the relaxation step, the kernel is simply applied on the pheromone field `map`. To apply a convolution with `ConvolveOp`, two images have to be given, and the convolved image is saved in `newmap`. Since the updated map is now `newmap`, the two variables are swapped.

The full update of the slime simulation is done in this short function:

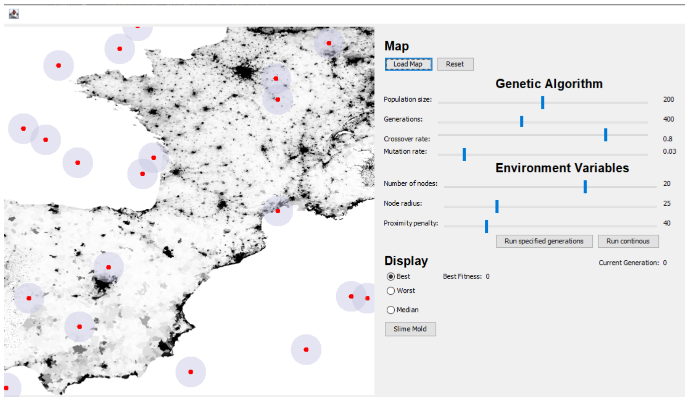
```
void updateMap() {
    updateAgents();
    propagatePheromones();
    relaxationStep();
}
```

Each agent is first updated, their pheromones are propagated and the relaxation step is performed.

5.4. Assessment

This section showcases the program built as part of this BSP, and presents some difficulties encountered during the development of this work. Possible improvements are then listed, which were not implemented within this BSP due to the limited time frame.

5.4.0.1. Program Presentation. The program developed runs in Java JDK 11, and can be run by either launching the `BspNetworkGen-1.0-SNAPSHOT.jar` in the folder `target`, or by opening the project in Netbeans and launching it from there. After launching the program, this user interface is presented to the user:

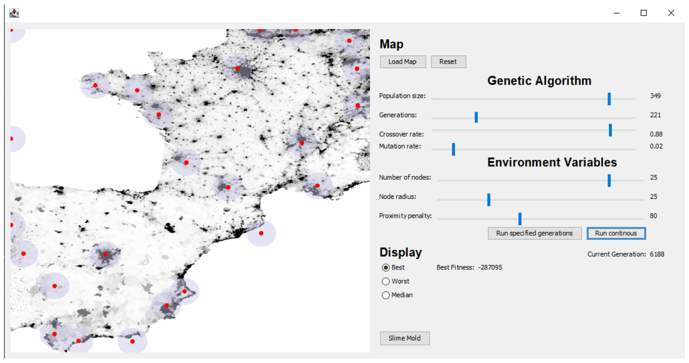


Population size: 200
Generations: 400
Crossover rate: 0.8
Mutation rate: 0.03
Number of nodes: 20
Node radius: 25
Proximity penalty: 40
Current Generation: 0
[✓ Best] Best Fitness: 0
Worst
Median

Fig. 7: The first menu of the program

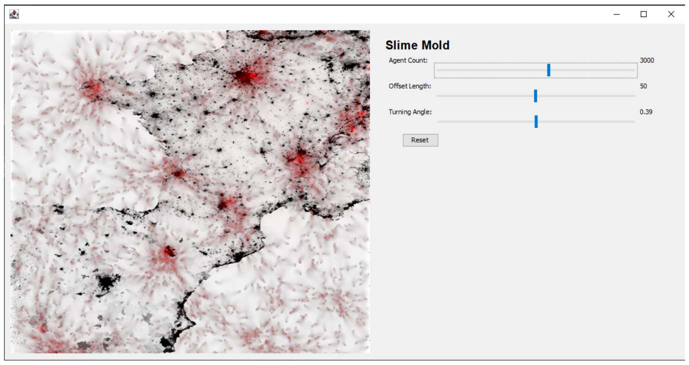
The population density map as well as a solution as overlay are displayed on the left. Which solution is displayed can be changed using the radio button under Display. Many parameters can be changed on the right with sliders, such as the parameters for the genetic algorithm and for the environment. When changing the node radius, the program may take a few seconds since it needs to recompute the reach for each point. At the top, the population density map image can be changed. It may be necessary to re-scale this image to be of size 600x600 because of limitations of the program.

The *Reset* button, starts the genetic algorithm with a new set of agents. To run the genetic algorithm, one can either choose to run exactly the number of generations n as adjusted with the slider with "*Run specified generations*", or run the generations continously, and update the displayed solution every n generations.



Population size: 349
Generations: 221
Crossover rate: 0.88
Mutation rate: 0.02
Number of nodes: 25
Node radius: 25
Proximity penalty: 80
Current Generation: 6188
[✓ Best] Best Fitness: -287095
Worst
Median

When the user is satifised with the solution obtained, the button "*Slime Mold*" sends the user to the interface to perform the slime mold simulation. When opening the program, the agents are spread around the map, but they rapidly create interesting shapes.



Agent Count: 3000
Offset Length: 50
Turning Angle: 0.39

Fig. 8: The menu for the slime mold simulation

Here are some of interesting shapes that can be obtained when changing the parameters:

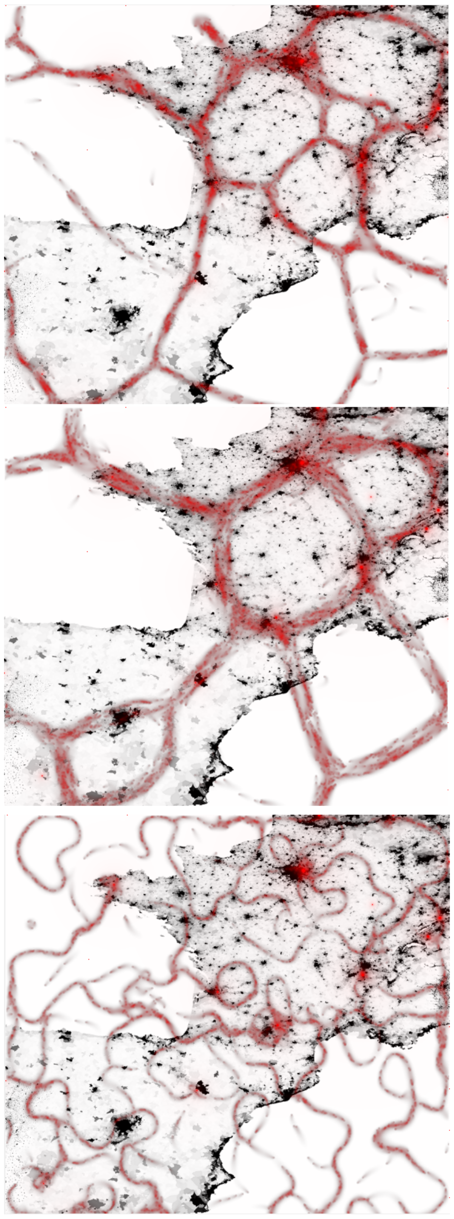


Fig. 9: Shapes obtained with decreasing offset length. Note, the second image is very similar to the France high speed rail network, if some parts of the slime mold network are omitted.

5.4.0.2. Difficulties encountered.. During the development of the project two main issues were encountered. The first issue was only found when analysing the results produced of the scientific deliverable. This issue was a bad implementation of the crossover function in the genetic algorithm. This issue did not take long to resolve, but took time to find. The other issue is described in the production part, under the description of the implementation of the slime mold. Here, some performance issues as and compromises with the data points occurred.

5.4.0.3. Possible improvements. There are quite a few possible improvements:

- Work with population density of any sizes
  At this current stage, the program basically only works flawlessly with images of size 600x600
- Saving and loading solutions
  It would be useful to be able to save and load solutions generated with the genetic algorithm as well as with the slime mold. Currently, the only way to do that is to take screenshots.
- Make the slime mold adapt to the terrain
  It would be possible, given a topological map, to make the slime mold avoid these points. This would be a task of higher difficulty.

5.4.0.4. Overall Assessment. Overall, the program achieves the requirements, but each step is not refined. This program would not have any real-world benefits, unless much further improvements are done. However, as a project done as part of a BSP, this was very successful and managed to touch many different topics (genetic algorithms, user interface creation, image manipulation, swarm simulations,...) in an educating manner.

Acknowledgment

I would like to thank my tutor, Grégoire Danoy, for his support and help through the semester.

6. Conclusion

This BSP contains the production of two deliverables. A first scientific deliverable, where a sensitivity analysis was performed on a genetic algorithm that tries to optimise some model. This model was also described as part of this deliverable, and represented the positioning of points on a population density such that they cover the most population whilst keeping distance of eachother. The second deliverable is the program implemented the genetic algorithm and the model, as well as a slime mold simulation that used the areas of high population density to produce a network connecting these points.

As the author of this paper, I wish to express my interest towards the fields touched within this BSP, and share that working on this project was a highly education and valuable experience.

7. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a precheck by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person's work without acknowledging the source
  6) Paraphrasing another person's work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
  8) Using another person's unpublished work without attribution and permission ('stealing')
  9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

  Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[BiCS(2021)] BiCS Bachelor Semester Project Report Template. https://github.com/nicolasguelfi/lu.uni.course.bics.global University of Luxembourg, BiCS - Bachelor in Computer Science (2021).

[BiCS(2021)] Bachelor in Computer Science: BiCS Semester Projects Reference Document. Technical report, University of Luxembourg (2021)

[1] Cacuci, Dan G.; Ionescu-Bujor, Mihaela; Navon, Michael (2005). Sensitivity and Uncertainty Analysis: Applications to Large-Scale Systems. Vol. II. Chapman & Hall.

[2] Gridded Population of the World, v4.11 (2020), 1km resolution Socioeconomic Data and Applications Center (SEDAC), NASA's Earth Observing System Data and Infromation System (EOSDIS) Hosted by CIESIN, Columbia University

[3] Tero, Atsushi & Takagi, Seiji & Saigusa, Tetsu & Ito, Kentaro & Bebber, Daniel & Fricker, Mark & Yumiki, Kenji & Kobayashi, Ryo & Nakagaki, Toshiyuki. (2010). Rules for Biologically Inspired Adaptive Network Design. Science (New York, N.Y.). 327. 439-42. 10.1126/science.1177894.

[4] Monte Carlo Physarum Machine: Characteristics of Pattern Formation in Continuous Stochastic Transport Networks Oskar Elek, Joseph N. Burchett, J. Xavier Prochaska, Angus G. Forbes, University of California, Santa Cruz, New Mexico State University, Kavli IPMU

[5] Genetic Algorithm for Solving Simple Mathematical Equality Problem, Denny Hermawanto, Indonesian Institute of Sciences (LIPI)

[6] METAHEURISTICS, From Design To Implementation El-Ghazali Talbi, University of Lille – CNRS – INRIA, John Wiley & Sons Inc.

8. Appendix